Hundhausen, C. et al. (Eds.) (2010). *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Leganés-Madrid, Spain 21-25 Septemebr 2010: Proceedings.* IEEE, 127-130.
DOI: http://dx.doi.org/10.1109/VLHCC.2010.26

# Lightweight Executability Analysis of Graph Transformation Rules

Elena Planas
Univ. Oberta de Catalunya
eplanash@uoc.edu

Jordi Cabot
INRIA - École des Mines de Nantes
jordi.cabot@inria.fr

Cristina Gómez
Univ. Politècnica de Catalunya
cristina@essi.upc.edu

Esther Guerra
Univ. Carlos III de Madrid
eguerra@inf.uc3m.es

Juan de Lara
Univ. Autónoma de Madrid
Juan.deLara@uam.es

***Abstract*—Domain Specific Visual Languages (DSVLs) play a cornerstone role in Model-Driven Engineering (MDE), where (domain specific) models are used to automate the production of the final application. Graph Transformation is a formal, visual, rule-based technique, which is increasingly used in MDE to express in-place model transformations like refactorings, animations and simulations. However, there is currently a lack of methods able to perform *static* analysis of rules, taking into account the DSVL meta-model integrity constraints.**

**In this paper we propose a lightweight, efficient technique that performs static analysis of the *weak executability* of rules. The method determines if there is some scenario in which the rule can be safely applied, without breaking the meta-model constraints. If no such scenario exists, the method returns meaningful feedback that helps repairing the detected inconsistencies.**

## I. Introduction

Domain Specific Visual Languages (DSVLs) play a fundamental role in Model-Driven Engineering (MDE), where they are used to represent domain-specific concepts and expert knowledge in both vertical and horizontal domains. Their value dramatically increases when combined with powerful code generators, so that (domain-specific) models are no longer passive documentation, but actively used to produce most of the code of the final application.

Graph Transformation (GT) [2] is a formal, declarative, rule-based technique for expressing model manipulations. It is gaining popularity in MDE due to its visual nature, which makes rules intuitive. Hence, GT is used to complement meta-models (which describe the DSVL syntax) for expressing the *behaviour* of the DSVLs. At the same time, the formal basis of GT makes rules analysable, and thus, facilitates the verification of the DSVL definition. However, even though GT offers a rich body of theoretical results [2], [7], there is still further work to be done regarding its integration with MDE techniques. In particular, there is a lack of methods to analyse rule correctness with respect to the DSVL meta-model and its integrity constraints.

In order to alleviate this situation, we propose a lightweight, efficient, static analysis method to check the *weak executability* of rules. This is a basic correctness property that studies whether there is a possible scenario where a rule can be safely applied without breaking any meta-model constraint. Our method translates the rule to be analysed into operational actions, and checks the dependencies between these actions and the structural constraints in the meta-model (e.g. cardinalities). For each detected error, it suggests possible corrections to make the rule weakly executable, expressed in the form of modified rules or as graphical patterns that the rule cannot contain. The method works at design time without any need to execute the rules or synthesize test scenarios.

**Paper Organization.** Section II introduces meta-modeling and GT. Section III shows the translation of rules into actions. Sections IV and V explain how to check rule executability and build the feedback. Section VI discusses related work and Section VII concludes.

## II. Syntax and Semantics of DSVLs

In this section we present a DSVL for production systems that we will use to illustrate our method.

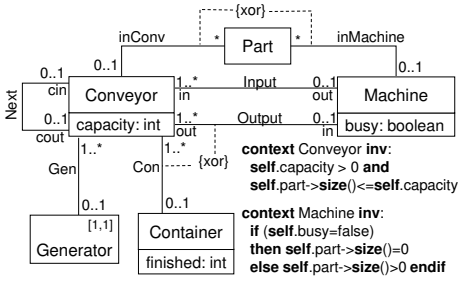


Figure 1. DSVL meta-model.

Fig. 1 shows the meta-model that defines the syntax of the DSVL. It includes elements of type conveyor that can be connected to other conveyors, to generators of parts, to containers or to machines. Conveyors can contain parts up to its maximum capacity (attribute *capacity*), which is

controlled by the OCL invariant in class *Conveyor*. Parts can either be transported in a conveyor or processed in a machine, but not simultaneously. Containers are terminal elements that count the number of parts that have finished. Production systems must contain exactly one generator, and we show this restriction in the top-right corner of class *Generator*.

In order to specify the DSVL operational semantics, we use GT as it allows using the concrete syntax of the DSVL in the rules, making them intuitive for the designer. A graph grammar is made of a set of rules and an initial graph (*host graph*) to which the rules are applied. Each rule is made of a left and a right hand side (LHS and RHS) graph. The LHS expresses pre-conditions for the rule to be applied, whereas the RHS contains the rule's post-conditions. In order to apply a rule to a *host graph*, a morphism (an occurrence or *match*) of the LHS has to be found in it. Then, the rule is applied by substituting the match by the rule's RHS. The grammar execution proceeds by applying the rules to the host graph as long as possible in non-deterministic order.

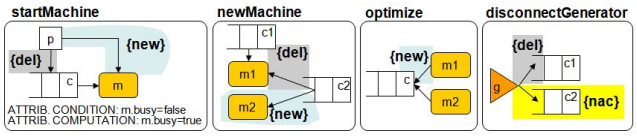


Figure 2. Some rules of the DSVL.

Fig. 2 shows some rules describing the DSVL operational semantics. Even though rules are declarative, we use a compact and operational notation. The elements created by the rules are enclosed in a polygon labeled `new`, while the elements deleted by the rules are labeled `del`. Rule "startMachine" starts the processing of a part (depicted as a white square) taking out it from the conveyor (lattice box), putting it into a free machine (colored square), and changing its state to busy (the pre and post value of this attribute is controlled by the attribute condition and computation sections). Rule "newMachine" incorporates a new machine to the plant, receiving parts initially processed by existing overloaded machines. Rule "optimize" maximizes the use of conveyors by allowing two machines to share them as output, whereas rule "disconnectGenerator" disconnects a generator (triangle). This latter rule uses a *Negative Application Condition* (NAC), a pattern that is forbidden to occur for the rule to be applicable.

## III. Deriving Actions from GT Rules

Prior to check the executability of a rule, our method performs a pre-processing step to translate the rule into an action-based representation that captures the structural changes the rule produces on the host graph. Hence, given a rule and the DSVL meta-model, we express the rule effects as a list of the following actions:

- *CreateObject(t:Type):Object*: Creates and returns a new object that conforms to the type $t$.
- *DestroyObject(o:Object)*: Destroys the object $o$.
- *UpdateAttribute(o:Object, attr:Attribute, v:Value)*: Sets $v$ as the new value for the attribute $attr$ of $o$.
- *CreateLink(as:Association, $r_a$:Role, $o_1$:Object, $r_b$:Role, $o_2$:Object):Link*: Creates and returns a new link (i.e. association instance) in the binary association $as$ between objects $o_1$ (role $r_a$) and $o_2$ (role $r_b$).
- *DestroyLink(as:Association, $r_a$:Role, $o_1$:Object, $r_b$:Role, $o_2$:Object)*: Destroys the link between the objects $o_1$ (role $r_a$) and $o_2$ (role $r_b$) from the association $as$.

In particular, the effect of a rule consists of creating the elements labelled `new`, deleting the elements labelled `del`, and updating the attributes according to the attribute computation section. NACs and attribute conditions are not considered since they do not produce modification effects on the host graph.

For instance, the actions derived from rule "startMachine" are the following:

**Rule** "startMachine"
(1) l := CreateLink(inMachine,part,p,machine,m)
(2) DestroyLink(inConv,part,p,conveyor,c)
(3) UpdateAttribute(m,busy,true)

The first action creates the link between the part $p$ and the machine $m$; the second action deletes the link between $p$ and the conveyor $c$; and the last action tags the machine as busy.

## IV. Executability of GT Rules

A rule $r$ is *weakly executable* if it has a chance of being successfully executed. That is, if we can find at least one host graph $G$ on which $r$ can be applied and the direct derivation $G \Rightarrow^r H$ generates a graph $H$ consistent with the system's integrity constraints. Otherwise $r$ is useless, as every time it is executed, an error arises because $H$ violates some integrity constraint.

Rules may fail to be weakly executable when their post-conditions do not take into account the possible dependencies between the actions performed by the rule on the host graph. Due to the integrity constraints some actions may require executing other actions as part of the same rule in order to leave the system in a consistent state.

Note that we define our *executability* property as *weak executability* since we do not require all executions of the rule on a matching host graph to be successful, which could be defined as *strong executability*. Weak executability is a prerequisite for strong executability (the latter implies the former). Hence, designers can check first weak executability, which is simpler to verify, and then they can apply other techniques to determine the stronger property if necessary (see Section VI).

As an example, rule "newMachine" is not executable since, every time we create a new machine and do not associate it to any output conveyor, we reach an erroneous state where the minimum 1 cardinality of the $Output$ association on the role $out$ is violated. Instead, rule "startMachine" is weakly executable since we are able to find an execution scenario where we can successfully move a part from a conveyor to a machine.

In order to check the weak executability of a rule we proceed by first deriving the modification actions it performs and then applying a two-step process sketched in Fig. 3. Both steps are done at design-time, taking into account only the definitions of the meta-model and the rules. Moreover, our method is *static*, improving its efficiency.

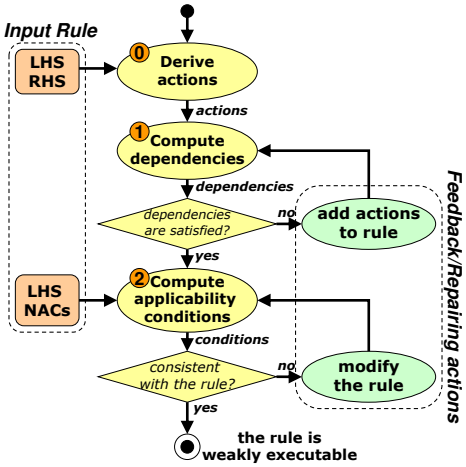


Figure 3. Overview of the process.

### A. Step 1: Dependencies

The executability of a rule depends on the actions it performs. In particular, problems may arise when some action requires the presence of other actions within the same rule in order to reach a consistent state after executing the rule. Therefore, to be executable, a rule $r$ must satisfy all dependencies for every action $ac$ in $r$. Dependencies for a specific action are drawn from the structure and constraints of the DSVL meta-model and from the kind of modification the action performs. It may happen that an action depends on several actions or that we have different alternatives to keep the consistency after executing it. In the latter case, the dependency is satisfied as long as one of the possible dependee actions appears. The dependencies shown in this paper are an adaptation of the ones appearing in [5].

For example, as shown in top of Fig. 4, rule "newMachine" has three mandatory dependencies. They express that every time we create a new machine we must link it with an $input$ and an $output$ conveyor ($dep_{1.1}$ and $dep_{1.2}$) to avoid violating the minimum cardinality "1" of association $Input$ and $Output$ respectively, as well as update the value of its attribute $busy$ ($dep_{1.3}$). The rule "newMachine" does not include some of these changes and, hence, it is not executable.

In contrast, actions of rule "startMachine" (down of Fig. 4) have several alternative dependencies. In particular, the creation of the link between the part and the machine (action labeled "1") has three disjoint dependencies: taking out the part from another machine ($dep_{1.1}$), taking out the part from a conveyor ($dep_{1.2}$) or creating the part itself ($dep_{1.3}$). The deletion of the link between the part and the conveyor (action labeled "2") has also three disjoint dependencies: putting the part in a conveyor ($dep_{2.1}$), putting the part in a machine ($dep_{2.2}$) or deleting the part ($dep_{2.3}$). The rule satisfies one of the disjoint sequences of dependee actions ($dep_{1.2}$ and $dep_{2.2}$), hence, it is weakly executable.

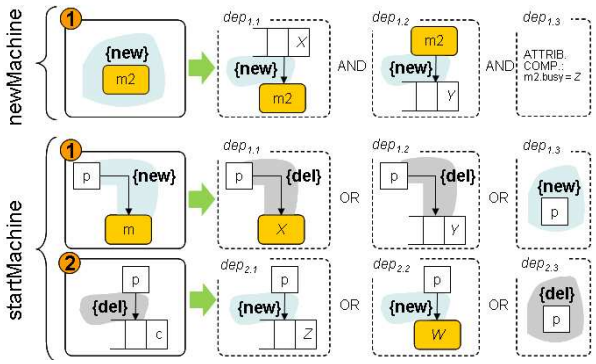


Figure 4. Dependencies for rules "newMachine" and "startMachine".

Rules that do not satisfy some dependency can be extended by adding the missing actions. This extension is a necessary condition but not sufficient to guarantee the executability of the rule, as the added actions may introduce new dependencies. Thus, after extending a rule, it must be checked again. Dependencies for the added actions can be satisfied by previous existing actions.

### B. Step 2: Applicability conditions

As a result of the previous step we know whether there are some host graphs that satisfy the meta-model constraints after applying the rule. This second step characterizes those host graphs and checks that at least one of them can actually be a match for the rule considering its LHS and NACs. Otherwise, the rule is useless since it can never be applied on graphs that would satisfy the meta-model constraints at the end of the rule execution.

For instance, the rule "optimize" is not weakly executable since the single scenario in which the rule can be successfully executed (the one in which the conveyor $c$ is not the output of any machine before executing the rule) is forbidden by its LHS (which forces the conveyor to be the output of machine $m2$ in order to be a match for the rule).

Applicability conditions are defined as anti-patterns (i.e. a kind of graph constraints [2]) expressing conditions that are not allowed to be found in the rule, as otherwise those patterns would forbid the match that makes the rule executable. In the rule "optimize", the anti-pattern would state that the LHS cannot contain a graph pattern including a link of *Output* between *c* and a machine.

## V. Feedback

Our method processes the results of the previous verification steps to provide a more amenable feedback, graphically expressed using the same DSVL syntax used in the rules.

The first step of our process returns the possible dependee actions that should be added to the rule in order to make it executable. We translate back these missing dependencies, incorporating them into the semantics of the original rule. As an example, Fig. 5 shows to the left a new version of rule "newMachine", now weakly executable since the added actions ensure the rule satisfies its dependencies.

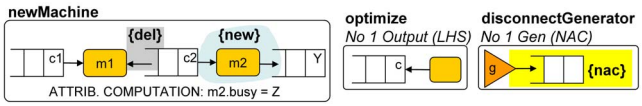


Figure 5. Feedback.

The second step of our process returns the set of patterns that are not allowed to be found in the LHS or NACs of the rule. This feedback is presented to the user as a graphical pattern that is forbidden to occur either in the LHS or the NAC, but which may include some initially bound elements. For example, the center of Fig. 5 shows a forbidden pattern for rule "optimize" (the *c* node in the pattern is bound to the *c* node in the LHS). To make the rule executable, designers have to modify the rule in order to avoid any occurrence of this pattern in it. Similarly, the right part of the figure shows a forbidden pattern for rule "disconnectGenerator" (according to the DSVL constraints, a generator must be connected with at least one conveyor).

## VI. Related Work

GT has developed a number of analysis techniques [2], [3], [4], [7] which however do not consider the meta-model integrity constraints. There are some efforts to integrate GT with meta-modeling, though. In [8], graph constraints are derived from restricted sets of meta-model integrity constraints, which can be used to generate pre-conditions for the rules, ensuring strong executability. Our goal instead is different, as we analyse rules to detect (the lack of) weak executability.

Other methods are based on the enumeration of all possible conflict contexts [2], on constraint solving techniques [1] or on model checking [6]. In contrast, our method performs a static analysis of the rules and hence it is more efficient. As a trade-off, these other methods can verify more complex correctness properties. Therefore, both approaches can be combined depending on the property to verify.

With respect to the used techniques, they have also been exploited to verify specifications in other domains, specifically, to verify operations defined by means of Action Semantics in the UML context [5].

Finally, an important difference with respect to the previous works is that our feedback is given visually, minimizing the need of interpretation by the designer.

## VII. Conclusions and Further Work

In this paper we have presented a method for the design-time analysis of the executability of GT rules.

The method provides feedback enabling the semi-automatic correction of the rules, or visually highlighting the problems detected in their LHS and NACs.

In the future we would like to cover new properties like completeness or redundancies and to give as feedback the possible modifications in the meta-model (and not only in the rule) that would make the rule executable. We also plan to provide tool support, integrating it into a tool for GT.

## VIII. Acknowledgements

Work funded by the Spanish Ministry of Science and Innovation through mobility grants JC2009-00015 and PR2009-0019, projects TIN2008-02081 and TIN2008-00444 and the R&D programme of the Madrid Community, project S2009/TIC-1650.

## References

[1] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. A UML/OCL framework for the analysis of graph transformation rules. *Software and Systems Modeling*, 9(3):335, 2010.

[2] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.

[3] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

[4] L. Lambers, H. Ehrig, and G. Taentzer. Sufficient criteria for applicability and non-applicability of rule sequences. *ECEASST*, 10, 2008.

[5] E. Planas, J. Cabot, and C. Gómez. Verifying action semantics specifications in UML behavioral models. In *CAiSE*, volume 5565 of *LNCS*, pages 125–140. Springer, 2009.

[6] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *ICGT*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.

[7] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[8] G. Taentzer and A. Rensink. Ensuring structural constraints in graph-based models with type inheritance. In *FASE*, volume 3442 of *LNCS*, pages 64–79, 2005.